THIS FILE CONTAINS PANDAS FOR DATA ANALYTICS - BY RUPAM GUPTA

Python with Pandas is used in a wide range of fields including academic and commercial domains including finance, economics, Statistics, analytics, etc

Key Features of Pandas

```
1) Fast and efficient DataFrame object with default and customized indexing.
2) Tools for loading data into in-memory data objects from different file formats.
3) Data alignment and integrated handling of missing data.
4) Reshaping and pivoting of date sets.
5) Label-based slicing, indexing and subsetting of large data sets.
6) Columns from a data structure can be deleted or inserted.
7) Group by data for aggregation and transformations.
8) High performance merging and joining of data.
9) Time Series functionality.
```

```
•Data Analysis:-

Raw data - information- Prepare- Feature selection- Model Data
import data(Data Acquistion) - Data prepartion(Cleaning data, Data Engineer) - EDA - Model Data
```

Pandas deals with the following three data structures −

1) Series: Series is a one-dimensional array like structure with HOMOGENEOUS data.

2) DataFrame: DataFrame is a two-dimensional array with HETEROGENEOUS data

3) Panel: Panel is a three-dimensional data structure with heterogeneous data. It is hard to represent the panel in graphical representation. But a panel can be illustrated as a container of DataFrame. These data structures are built on top of Numpy array.(hardly used in the industry)

•Series (1D) , rows.

•Data Frames (2D) , rows and columns.

•Panel (3D).

In [1]:
```python
print("hello")
```

```
hello
```

if pandas is not installed then please the following code to install it.

pip install pandas

In [2]:
```python
#Its a good practice to import both numpy and pandas together.

import pandas as pd
import numpy as np
```

• The Pandas Series is much more general as well as flexible as compared to 1-D NumPy array that it emulates

-------------------------Series as generalized NumPy array-------------

• The Series object is basically interchangeable with a 1-D NumPy array

• The significant difference is the presence of the index: whereas the Numpy Array has an implicitly defined integer index used in order to obtain the values, the Pandas Series has a clear-cut defined index associated with the values

• The Series object additional capabilities are provided by this clear index description.

The index needs not to be an integer but can made up of values of any wanted type. For instance, we can use strings as an index:)

In [3]:
```python
(1)#Trying out Series data structure. In the output the values will be float as we mentioned dtype=float, a=tuple & b=list.

a=(1,2,3,4,5,6)
b=["one", "two", "three", "four", "five", "six"]

x=pd.Series(a,index=b,dtype=float)
x
```

Out[3]:
```
one      1.0
two      2.0
three    3.0
four     4.0
five     5.0
six      6.0
dtype: float64
```

In [4]:
```python
(2)#Trying out Series data structure. In the output the values will be string as we mentioned dtype=str,a=tuple & b=list.

a=(1,2,3,4,5,6)
b=["one", "two", "three", "four", "five", "six"]

y=pd.Series(b,index=a,dtype=str)
y
```

Out[4]:
```
1      one
2      two
3    three
4     four
5     five
6      six
dtype: object
```

The Pandas Series Object

• A Pandas Series is a 1-D array of indexed data. It can be created from a array or list as shown in the following code:

• If don't use index command then pandas will start the indexing from "0" automatically. Lets see below:-

In [5]:
```python
(3)#Series without index command, we will see the 1st column from the result start with "0"

d_series1= pd.Series([0.1,0.15,0.20,0.25])
d_series1
```

Out[5]:
```
0    0.10
1    0.15
2    0.20
3    0.25
dtype: float64
```

As shown in the output, A sequence of indices and sequence of values both are wrapped by the series, which we can access with the index attributes and values. The values are simply a familiar NumPy array:

In [6]:
```python
d_series1.values
```

Out[6]:
```
array([0.1 , 0.15, 0.2 , 0.25])
```

In [7]:
```python
d_series1.index
```

Out[7]:
```
RangeIndex(start=0, stop=4, step=1)
```

As NumPy array, data can be obtained by the associated index through the familiar Python square-bracket notation:

In [8]:
```python
d_series1[3] #we can extract the elements from the series using the square braket.
```

Out[8]:
```
0.25
```

In [9]:
```python
d_series1[:3] #returns along with the index
```

Out[9]:
```
0    0.10
1    0.15
2    0.20
dtype: float64
```

In [10]:
```python
d_series1[:-2] #return elements after -2 position i.e(after 0.2)
```

Out[10]:
```
0    0.10
1    0.15
dtype: float64
```

Creating series from dictionary

Series as specialized dictionary • A dictionary is a structure which maps arbitrary keys to a collection of arbitrary values, as well as a Series is a structure which maps typed keys to a set of typed values

• This typing is significant: just as the type-specific compiled code behind a NumPy array makes it more well-organized than a Python list for certain operations, the type information of a Pandas Series makes it much more efficient as compare to Python dictionaries for certain operations

In [11]:
```
#here we are creating a dictionary with keys & value.
#then creating a veriable i.e "details"
#The dictionary get converted to series, when we used "emp" as object.

emp ={"A":25,"B":30,"C":35,"D":40}
details = pd.Series(emp)
details
```

Out[11]:
```
A    25
B    30
C    35
D    40
dtype: int64
```

Note: Values are used by default as series elements & Keys as index

Dictionary is a mapping data type , We cannot manupulate index in as we do in case of List & Tuples.

In [12]:
```
city_dict = {'Delhi': 450014,
             'Mumbai': 787748,
             'Kolkata': 956225,
             'Chandigarh': 145247,
             'Chennai': 630063}
city_dict
```

Out[12]:
```
{'Delhi': 450014,
 'Mumbai': 787748,
 'Kolkata': 956225,
 'Chandigarh': 145247,
 'Chennai': 630063}
```

In [13]:
```
city_ser = pd.Series(city_dict) #converting dict to series
city_ser
```

Out[13]:
```
Delhi         450014
Mumbai        787748
Kolkata       956225
Chandigarh    145247
Chennai       630063
dtype: int64
```

• Array-style operations such as slicing is also supported by the Series:

• A Series will be built where the index is drawn from the sorted keys by default. Typical ictionary-style item access can be performed from here:

In [14]:
```
city_ser[3]
```

Out[14]:
```
145247
```

In [15]:
```
city_ser['Mumbai']
```

Out[15]:
```
787748
```

In [16]:
```
city_ser[3:]
```

Out[16]:
```
Chandigarh    145247
Chennai       630063
dtype: int64
```

In [17]:
```
city_ser[:3]
```

Out[17]:
```
Delhi      450014
Mumbai     787748
Kolkata    956225
dtype: int64
```

In [18]:
```
city_ser[:-1]
```

Out[18]:
```
Delhi         450014
Mumbai        787748
Kolkata       956225
Chandigarh    145247
dtype: int64
```

In [19]:
```
city_ser['Mumbai':]
```
Out[19]:
```
Mumbai        787748
Kolkata       956225
Chandigarh    145247
Chennai       630063
dtype: int64
```
In [20]:
```
city_ser[:'Mumbai']
```
Out[20]:
```
Delhi     450014
Mumbai    787748
dtype: int64
```
- **Data can be a scalar, which is repeated in order to fill the specified index:**

In [21]:
```
(1)
my_list = ['Rupam','Aman','Riya']
pd.Series('hello',index = my_list)
```
Out[21]:
```
Rupam    hello
Aman     hello
Riya     hello
dtype: object
```
In [22]:
```
(2)
my_list2 = ['a', 'b', 'c']
pd.Series(25,index = my_list2)
```
Out[22]:
```
a    25
b    25
c    25
dtype: int64
```
- **The index can be set explicitly in every case if a different result is preferred:**

In [23]:
```
(1)#here the index=[5,7] means it wants to extract values where keys are 5 & 7.

pd.Series({4:'w',5:'y',6:'u',7:'i'},index = [5,7]) ## customizable index output
```
Out[23]:
```
5    y
7    i
dtype: object
```
In [24]:
```
(2)#here the missing values is denoted as NaN=not a number

my_items ={'mobile':50000,'shirt':3000, 'shoes':5000, 'car':2500000}

pd.Series(my_items, index = ['mobile','car','bag'])
```
Out[24]:
```
mobile      50000.0
car       2500000.0
bag             NaN
dtype: float64
```
In [25]:
```
(3)#NAN

d = {'a': 1, 'b': 2, 'c': 3}
ser = pd.Series(d, index=['x', 'y', 'z'])
ser
```
Out[25]:
```
x   NaN
y   NaN
z   NaN
dtype: float64
```
Importing file to pandas, write pd.read_csv(file_path);file path could be from internal source or external source

In [34]:
```
import pandas as pd
RGdiamonds = pd.read_csv('https://raw.githubusercontent.com/mwaskom/seaborn-data/master/diamonds.csv')
```

In [35]:
```
RGdiamonds #returns the complete table
```

Out[35]:

	carat	cut	color	clarity	depth	table	price	x	y	z
0	0.23	Ideal	E	SI2	61.5	55.0	326	3.95	3.98	2.43
1	0.21	Premium	E	SI1	59.8	61.0	326	3.89	3.84	2.31
2	0.23	Good	E	VS1	56.9	65.0	327	4.05	4.07	2.31
3	0.29	Premium	I	VS2	62.4	58.0	334	4.20	4.23	2.63
4	0.31	Good	J	SI2	63.3	58.0	335	4.34	4.35	2.75
...	...	...	...	...	...	...	...	...	...	...
53935	0.72	Ideal	D	SI1	60.8	57.0	2757	5.75	5.76	3.50
53936	0.72	Good	D	SI1	63.1	55.0	2757	5.69	5.75	3.61
53937	0.70	Very Good	D	SI1	62.8	60.0	2757	5.66	5.68	3.56
53938	0.86	Premium	H	SI2	61.0	58.0	2757	6.15	6.12	3.74
53939	0.75	Ideal	D	SI2	62.2	55.0	2757	5.83	5.87	3.64

53940 rows × 10 columns

In [31]:
```
RGdiamonds.head() #returns only first 5 rows.
```

Out[31]:

	carat	cut	color	clarity	depth	table	price	x	y	z
0	0.23	Ideal	E	SI2	61.5	55.0	326	3.95	3.98	2.43
1	0.21	Premium	E	SI1	59.8	61.0	326	3.89	3.84	2.31
2	0.23	Good	E	VS1	56.9	65.0	327	4.05	4.07	2.31
3	0.29	Premium	I	VS2	62.4	58.0	334	4.20	4.23	2.63
4	0.31	Good	J	SI2	63.3	58.0	335	4.34	4.35	2.75

In [36]:
```
RGdiamonds.tail() #returns only last 5 rows.
```

Out[36]:

	carat	cut	color	clarity	depth	table	price	x	y	z
53935	0.72	Ideal	D	SI1	60.8	57.0	2757	5.75	5.76	3.50
53936	0.72	Good	D	SI1	63.1	55.0	2757	5.69	5.75	3.61
53937	0.70	Very Good	D	SI1	62.8	60.0	2757	5.66	5.68	3.56
53938	0.86	Premium	H	SI2	61.0	58.0	2757	6.15	6.12	3.74
53939	0.75	Ideal	D	SI2	62.2	55.0	2757	5.83	5.87	3.64

In [44]:
```
RGdiamonds.head(10) #returns first 10 rows
```

Out[44]:

	carat	cut	color	clarity	depth	table	price	x	y	z
0	0.23	Ideal	E	SI2	61.5	55.0	326	3.95	3.98	2.43
1	0.21	Premium	E	SI1	59.8	61.0	326	3.89	3.84	2.31
2	0.23	Good	E	VS1	56.9	65.0	327	4.05	4.07	2.31
3	0.29	Premium	I	VS2	62.4	58.0	334	4.20	4.23	2.63
4	0.31	Good	J	SI2	63.3	58.0	335	4.34	4.35	2.75
5	0.24	Very Good	J	VVS2	62.8	57.0	336	3.94	3.96	2.48
6	0.24	Very Good	I	VVS1	62.3	57.0	336	3.95	3.98	2.47
7	0.26	Very Good	H	SI1	61.9	55.0	337	4.07	4.11	2.53
8	0.22	Fair	E	VS2	65.1	61.0	337	3.87	3.78	2.49
9	0.23	Very Good	H	VS1	59.4	61.0	338	4.00	4.05	2.39

In [45]:
```
RGdiamonds.tail(10) #returns first 10 rows
```

Out[45]:

	carat	cut	color	clarity	depth	table	price	x	y	z
53930	0.71	Premium	E	SI1	60.5	55.0	2756	5.79	5.74	3.49
53931	0.71	Premium	F	SI1	59.8	62.0	2756	5.74	5.73	3.43
53932	0.70	Very Good	E	VS2	60.5	59.0	2757	5.71	5.76	3.47
53933	0.70	Very Good	E	VS2	61.2	59.0	2757	5.69	5.72	3.49
53934	0.72	Premium	D	SI1	62.7	59.0	2757	5.69	5.73	3.58
53935	0.72	Ideal	D	SI1	60.8	57.0	2757	5.75	5.76	3.50
53936	0.72	Good	D	SI1	63.1	55.0	2757	5.69	5.75	3.61
53937	0.70	Very Good	D	SI1	62.8	60.0	2757	5.66	5.68	3.56
53938	0.86	Premium	H	SI2	61.0	58.0	2757	6.15	6.12	3.74
53939	0.75	Ideal	D	SI2	62.2	55.0	2757	5.83	5.87	3.64

In [40]:
```
RGdiamonds.shape #returns total Rows & total columns
```

Out[40]:
```
(53940, 10)
```

In [43]:
```
RGdiamonds.describe() #returns interquartile range or 5 number summary
```

Out[43]:

	carat	depth	table	price	x	y	z
count	53940.000000	53940.000000	53940.000000	53940.000000	53940.000000	53940.000000	53940.000000
mean	0.797940	61.749405	57.457184	3932.799722	5.731157	5.734526	3.538734
std	0.474011	1.432621	2.234491	3989.439738	1.121761	1.142135	0.705699
min	0.200000	43.000000	43.000000	326.000000	0.000000	0.000000	0.000000
25%	0.400000	61.000000	56.000000	950.000000	4.710000	4.720000	2.910000
50%	0.700000	61.800000	57.000000	2401.000000	5.700000	5.710000	3.530000
75%	1.040000	62.500000	59.000000	5324.250000	6.540000	6.540000	4.040000
max	5.010000	79.000000	95.000000	18823.000000	10.740000	58.900000	31.800000

In [48]:
```
RGdiamonds.info() #returns complete information about the dataset
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 53940 entries, 0 to 53939
Data columns (total 10 columns):
 #   Column   Non-Null Count  Dtype  
---  ------   --------------  -----  
 0   carat    53940 non-null  float64
 1   cut      53940 non-null  object 
 2   color    53940 non-null  object 
 3   clarity  53940 non-null  object 
 4   depth    53940 non-null  float64
 5   table    53940 non-null  float64
 6   price    53940 non-null  int64  
 7   x        53940 non-null  float64
 8   y        53940 non-null  float64
 9   z        53940 non-null  float64
dtypes: float64(6), int64(1), object(3)
memory usage: 4.1+ MB
```

In [53]:
```
RGdiamonds.dtypes #returns all the dtypes of the features
```

Out[53]:
```
carat      float64
cut         object
color       object
clarity     object
depth      float64
table      float64
price        int64
x          float64
y          float64
z          float64
dtype: object
```

In [55]:
```
RGdiamonds.dtypes.unique() #returns all the distinct dtypes
```

Out[55]:
```
array([dtype('float64'), dtype('O'), dtype('int64')], dtype=object)
```

In [56]:
```
RGdiamonds.isnull() #returns all the dtypes of the features
```

Out[56]:

	carat	cut	color	clarity	depth	table	price	x	y	z
0	False	False	False	False	False	False	False	False	False	False
1	False	False	False	False	False	False	False	False	False	False
2	False	False	False	False	False	False	False	False	False	False
3	False	False	False	False	False	False	False	False	False	False
4	False	False	False	False	False	False	False	False	False	False
...	...	...	...	...	...	...	...	...	...	...
53935	False	False	False	False	False	False	False	False	False	False
53936	False	False	False	False	False	False	False	False	False	False
53937	False	False	False	False	False	False	False	False	False	False
53938	False	False	False	False	False	False	False	False	False	False
53939	False	False	False	False	False	False	False	False	False	False

53940 rows × 10 columns

In [57]:
```
RGdiamonds.isnull().sum() ## column wise count of null values
```

Out[57]:
```
carat      0
cut        0
color      0
clarity    0
depth      0
table      0
price      0
x          0
y          0
z          0
dtype: int64
```

In [59]:
```
RGdiamonds['carat'].isnull().sum() ## column wise count of null values
```

Out[59]:
```
0
```

In [60]:
```
RGdiamonds['cut'].unique() ## all the unique distinct values/categories from that column
```

Out[60]:
```
array(['Ideal', 'Premium', 'Good', 'Very Good', 'Fair'], dtype=object)
```

In [62]:
```
RGdiamonds['cut'].value_counts() ##counts of each category inside that column
```

Out[62]:
```
Ideal        21551
Premium      13791
Very Good    12082
Good          4906
Fair          1610
Name: cut, dtype: int64
```

In [ ]: